Multi-Resource Packing for Cluster Schedulers

Robert Grandl[1,2] Ganesh Ananthanarayanan[1,3] Srikanth Kandula[1]
Sriram Rao[1] Aditya Akella[1,2]
Microsoft[1], Univ. of Wisconsin, Madison[2], Univ. of California, Berkeley[3]

Abstract– Tasks in modern data-parallel clusters have highly diverse resource requirements along CPU, memory, disk and network. We present Tetris, a multi-resource cluster scheduler that *packs* tasks to machines based on their requirements of all resource types. Doing so avoids resource *fragmentation* as well as *over-allocation* of the resources that are not explicitly allocated, both of which are drawbacks of current schedulers. Tetris adapts heuristics for the multi-dimensional bin packing problem to the context of cluster schedulers wherein task arrivals and machine availability change in an online manner and wherein task's resource needs change with time and with the machine that the task is placed at. In addition, Tetris improves average job completion time by preferentially serving jobs that have less remaining work. We observe that fair allocations do not offer the best performance and the above heuristics are compatible with a large class of fairness policies; hence, we show how to simultaneously achieve good performance and fairness. Trace-driven simulations and deployment of our Apache YARN prototype on a 250 node cluster show gains of over 30% in makespan and job completion time while achieving nearly perfect fairness.

Categories and Subject Descriptors

C.2.4 [Computer-Communication Networks]: Distributed Systems

Keywords

Cluster schedulers; multi-dimensional packing; makespan; completion time; fairness

1. INTRODUCTION

To analyze large datasets, it has become typical to use clusters of machines to execute *jobs* consisting of many *tasks*. Jobs of many applications coexist on these clusters and their tasks have *diverse resource demands*. For instance, machine learning tasks are CPU-intensive while sort tasks are memory-intensive. Often, tasks are constrained on *multiple* resources, e.g., reduce tasks that are both memory- as well as network-intensive.

Given such diversity, we seek to build a cluster scheduler that *packs* tasks to machines based on their requirements on all the resources. Our objective in packing is to maximize the task throughput (or minimize makespan [1]) and speed up job completion. Packing is important in today's clusters due to *balanced* machine specifications (e.g., enough memory to be able to use all available disk drives and enough cross-rack network bandwidth [8, 13, 15] to use all the available CPU cores). Therefore, *any* of the resources—cores, memory, disk or network—can become fully used on a machine and prevent further tasks from being scheduled there.

Current schedulers neither pack tasks nor consider all their relevant resource demands. This results in *fragmentation* and *over-allocation* of resources, respectively. (i) Schedulers divide resources into slots (corresponding to some amount of memory and cores [5]) and offer the slots greedily to the job that is furthest from its *fair* share [3, 4, 12]. Such scheduling results in resource fragmentation, the magnitude of which increases with the number of resources being allocated [20].[2] (ii) Schedulers also ignore disk and network requirements of tasks [5, 12, 18]. When assigning tasks to machines, they only check that tasks' CPU and memory needs are satisfiable. Hence, they can schedule many network or disk-intensive tasks on the same machine. Such *over-allocation* leads to interference–disk seeks or network incast– that can sharply lower throughput. Also, over-allocation wastes resources. For example, when two tasks that can both use all of the available network bandwidth on a machine are scheduled together, they will take twice as long to finish. In doing so, they hold to their cores and memory and prevent other tasks that do not need the network from using them. Our analysis shows that, due to fragmentation and over-allocation of resources, the state-of-the-art schedulers in Facebook's and Bing's analytics clusters delay job completions and increase makespan by over 45%.

Multi-resource packing of tasks is analogous to multi-dimensional bin packing. Given balls and bins with sizes in $\mathcal{R}^d$, where d is the number of resources to be allocated, multi-dimensional bin packing assigns the balls to the fewest number of bins. Doing so maximizes the number of simultaneously scheduled tasks, thus minimizing makespan and improving job completion time. Even though multi-dimensional bin packing is known to be APX-Hard [27],[3] several heuristics exist [20]. However, they do not directly apply to our scenario. Whereas the heuristics consider balls of a fixed size, the resource demands of tasks (a) vary with time and based on the machine that they are placed at, and (b) are elastic, as in, tasks can function with less than their peak demand. Further, whereas the heuristics assume all balls are known apriori, a cluster scheduler has to cope with *online* arrivals of jobs, the *dependencies* between tasks, and cluster activity such as evacuation and ingestion of new data which compete for resources with tasks.

This paper presents Tetris, a cluster scheduler that packs tasks to machines based on their requirements along multiple resources. Tetris adaptively learns task requirements t_r and monitors available resources at machines m_r. The packing heuristic projects t_r and m_r into a euclidean space and picks the ⟨task, machine⟩ pair with the highest dot product value. Task requirements are adjusted to reflect placement at the given machine; only tasks whose requirements are satisfiable are considered; and the dot product prefers large tasks and those that use resources in proportions similar to what is available; for e.g., if machine has network free, other things being equal,

[1] Makespan = time to finish a set of jobs

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.

http://dx.doi.org/10.1145/2619239.2626334.

[2] With d resources, the packing efficiency can be $(\frac{1}{2d})^{\text{th}}$ of optimal.
[3] APX-Hard means that there is no asymptotic polynomial time approximation unless P=NP; it is a strict subset of NP-hard.

a network-intensive task has a higher dot product. This heuristic prevents over-allocation and reduces resource fragmentation.

Achieving good packing efficiency improves makespan but does not necessarily speed up individual jobs. Preferentially offering resources to the job with the *smallest remaining time* (SRTF) will minimize average job completion time [16]. However, the job with the least remaining work may not have tasks that pack well and hence a strict job time ordering can slow down everyone. Tetris develops a multi-resource version of SRTF for jobs which are DAGs of dependent tasks and combines both heuristics – best packing and shortest remaining job time – to reduce average job completion time.

While we have focused on performance metrics thus far, *fairness* is an important tool to ensure predictable performance. Prior work has developed multi-resource versions of fairness [12]. We show by counter-example as well as our evaluation that fair allocations, even when *pareto efficient* [4] and *work conserving*, do not yield the best job completion time and makespan. We also find that the best-performing schedule is not incompatible with the most-fair schedule. In particular, typical fair schedulers operate by offering resources to the job (or group of jobs) that is currently furthest from fair share. Using Tetris to pick the best-for-packing task from among this constrained subset will improve performance without compromising on fairness. Tetris offers a natural generalization— use the packing and job time heuristics to pick the best task from the $(1-f)$ fraction of running jobs (or groups) that are furthest from their fair share. Choosing $f = 0$ leads to best makespan and completion time, while $f \to 1$ offers *strict* fairness. We will show that for $f \in [.25, .5]$, Tetris offers nearly the best performance and the impact on jobs due to the resulting unfairness is negligibly small.

We believe our work makes the following advancements.

- We identify the importance of scheduling and packing *all relevant resources*. Otherwise, resources get fragmented and can be over-allocated, significantly affecting performance.
- We present a heuristic solution to the APX-hard problem of *packing* tasks along multiple resources in cluster schedulers.
- We show how to combine heuristics that improve packing efficiency with those that lower average job completion time.
- We show that pareto-efficient fair allocations do not yield best performance. Whereas performance and fairness are often unacheivable together, we show that in the context of cluster schedulers much better performance can be achieved with just a little unfairness and expose the trade-off with a knob.

We have built Tetris, the first data-parallel cluster scheduler to explicitly consider multi-resource packing, as a modification to the scheduler in YARN [5]. Tetris estimates task demands from previous executions of the same job and from completed tasks of the current job. Tetris uses a resource tracker to independently monitor the utilizations at the machines, thus allowing it to account for any errors in demand estimations and to adjust scheduling around unforeseen hotspots and misbehaving nodes. Tetris has been evaluated in simulations over production traces and using a deployment on a 250 machine cluster. Compared to state-of-the-art scheduling policies (slot-based [3, 4] as well as multi-resource fair (DRF) [12]), Tetris improves makespan and job completion by 30% in deployment and up to 40% in simulations over Facebook traces. These gains are 80% of the estimated gains from a simple upper-bound (not a true upper-bound since computing that is intractable). Further, Tetris offers a smooth trade-off between fairness and performance; we use $f = 0.25$ to achieve the above gains and fewer than 6% of the jobs slow down compared to a fair allocation; the average (max) slowdown is 6 (10)%.

[4] no job can increase share without decreasing the share of another

2. MOTIVATION

We motivate multi-resource packing of tasks using examples and production workloads. By devising an approximate upper bound on the potential gains from packing, we show that production jobs could speed up by 45% compared to existing schedulers.

2.1 Limitations of Existing Schedulers

Current schedulers are limited in their ability to pack tasks because they define slots based on only one resource (e.g., memory) and they allot slots to tasks based only on fairness. Job completion times and packing efficiency suffer as a consequence.

Slots: Analytics frameworks typically define slots based on only one resource (commonly, memory) [3, 4, 5, 18]. It is easy to see that statically sizing the slots leads to wastage and fragmentation when task requirements vary. While dynamic sizing of slots avoids wastage of the resource on which the slots are defined, other resources end up being *over-allocated*.

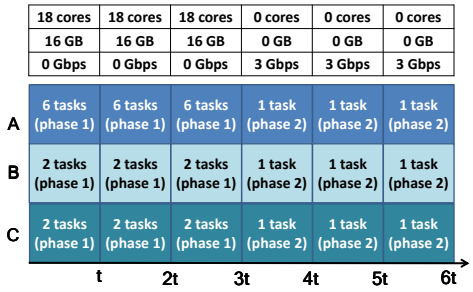
Take the example of intermediate tasks of a job (e.g., reduce tasks in MapReduce). These tasks read data from multiple machines and have high network requirements. Scheduling them based on only their memory footprint can cause needless contention for the network (and the disk if they write a lot of output). When tasks contend for a resource, the total effective throughput is lowered due to systemic reasons such as buffer overflows on switches (incast), disk seek overheads, and processor cache misses. Further, tasks delayed due to contention hold the memory for much longer and prevent other tasks from being scheduled at that machine.

Fairness: As mentioned earlier, a commonly used scheduling approach is to pick tasks from the job that is farthest from its fair share. A common problem with fairness based schedulers is that they do not *pack* the different resources. We use examples of Dominant Resource Fairness (DRF) [12] to illustrate this.

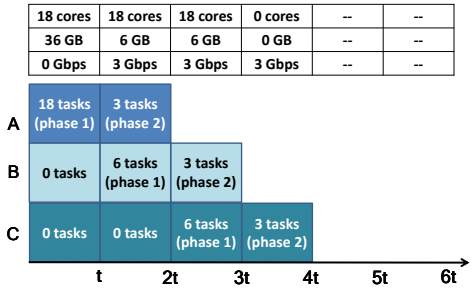
Consider a cluster with 18 cores, 36GB of memory and 3Gbps of network. Three jobs A, B and C have two phases each that are separated by a barrier; the "map" phase consists of 18, 6, and 6 tasks respectively and the "reduce" phase has 3 tasks for all jobs. Map tasks of job A require 1 core and 2GB of memory, while those for jobs B and C require 3 cores and 1GB of memory. All reduce tasks need 1Gbps of network and very little CPU or memory. Assume that all tasks run for t time units. Note that this example resembles map-reduce; map tasks are CPU and memory intensive while reduce tasks are network-intensive.

DRF will schedule 6 map tasks of job A and 2 map tasks each of jobs B and C, at a time, giving each job a dominant resource share of $\frac{1}{3}$ (A's dominant resource is memory whereas for B and C it is CPU cores). Hence, all of the map phases finish at $3t$. Such an allocation, however, leaves 20GB of cluster memory idle. For the reduce phase, DRF will run 1 reduce task from each of the jobs (network is the dominant resource for all jobs now) for a dominant resource share of $\frac{1}{3}$ per job. All jobs finish at 6t.

Consider a packing scheduler instead. Scheduling all 18 map tasks of A would fully use up the cluster's memory. Then, reduce tasks of A become runnable and have *complementary* resource demands to the mappers, allowing the cluster to schedule all 6 maps of B along with the 3 reducers of A. Such a schedule is shown in Figure 1. The jobs now finish at times 2t, 3t and 4t. Average completion time reduces by 50% over DRF (6t → 3t); the cluster's makespan reduces by 33% (6t → 4t) and every job finishes earlier compared to the fair schedule!



(a) Job schedule under DRF allocation



(b) Job schedule with multi-resource Packing

Figure 1: Comparing schedules for DAGs of tasks that result from fairness based allocation (e.g., DRF [12]) with alternatives. Each job has two phases with a strict barrier between them. The tables on top (in white) show resource utilization. Packing can avoid resource fragmentation and leverage complementarity of task requirements.

Observe that these gains hold if the packing schedule is used for any permutation among jobs A, B and C. Second, we implicitly extended DRF to also consider the network above; if DRF only considers CPU and memory, after the map phase phases, it would schedule all of the reduce tasks simultaneously since reducers have insignificant CPU and memory needs. Each reduce task then gets only $\frac{1}{3}$Gbps due to contention, causing tasks to run for 3t each and potentially longer if incast happens. Third, note that treating the cluster as one big bag of resources hides the impact of fragmentation. If instead the cluster consists of three machines with one-third cluster resources each, then with DRF all of the jobs will finish at 7t[5]. This is 43% worse makespan and 57% worse avg. completion time. Finally, the gains for the packing schedule accrue from avoiding fragmentation, preferring jobs with less remaining work and by exploiting complementarity in tasks' resource demands; we will show how Tetris show can do all of these.

The above examples highlight that fairness-based schedulers do not optimize job completion time. We have also discussed how slot-based scheduling causes fragmentation and wastage. And, disregarding relevant resources leads to their over-allocation and detrimental interference.

2.2 Workload Analysis

We build upon the aspects highlighted in the previous section to analyze their impact on production workloads.

[5]Here is the DRF schedule for the map phases; read it as the first map task of job A, A^1_m, scheduled at machine M_1 for the first t units. The reduce phases take an extra 3t similar to the case of Fig. 1(a).

	t	$2t$	$3t$	$4t$
M_1	$A^1_m A^2_m B^2_m A^7_m$	$C^2_m A^{10}_m A^{11}_m A^{12}_m$	$A^{13}_m A^{14}_m B^6_m$	C^6_m
M_2	$B^1_m A^3_m A^5_m A^8_m$	$B^3_m B^4_m$	$B^5_m A^{15}_m A^{17}_m$	
M_3	$C^1_m A^4_m A^6_m A^9_m$	$C^3_m C^4_m$	$C^5_m A^{16}_m A^{18}_m$	

	Bing	Facebook
Dates	12/2013, two days	1/2012, 1month
Cluster Size	Thousands	3000
Framework	Dryad [17]	Hadoop [1]
Script	Scope [30]	Hive [24]
DAG Depth	Large	2
File System	Cosmos	HDFS
Network Links	10Gbps	1Gbps
Oversubscription	<2	4

Table 1: Summary of datasets.

	Cores	Memory	Disk	Network
Cores	—	0.33, 0.41	0.22, 0.12	0.29, 0.23
Memory	—	—	−0.11, 0.28	0.04, −0.1
Disk	—	—	—	0.26, −0.07
Network	—	—	—	—

Table 2: Correlation matrix of task resource demands; each table entry shows the numbers for {Bing, Facebook}. There is little correlation across demands for various resources.

2.2.1 *Production Clusters*

We analyze production traces from Facebook's Hadoop and Bing's Cosmos clusters. These clusters consist of thousands of machines that are well provisioned in the number of cores, memory, disks and network connectivity. The clusters co-locate computation and storage. Table 1 lists some details. Both traces contain task-level information: start and end times, size of input and output, the machine on which it executed, along with tasks' resource requirements of CPU cycles, memory, disk IOPS and network bandwidth usage.

The computation frameworks in these clusters define slots based on cores and memory and *allot* slots to jobs based on fairness, i.e., distributing proportionally the slots between the running jobs. Both clusters preferentially place tasks close to their input data.

2.2.2 *Task Profiles*

Tasks are significantly diverse in their requirements. Figure 2 shows the variation of resource requirements of tasks across resources. CPU usage varies all the way from only 2% of a core to over 6 cores. Similarly, memory usage ranges from 100MB to nearly 17GB. While some tasks are IO-intensive, others are less so. Overall, the minimum values of resource requirements are 5−10× lower than the median, which in turn is 20× lower than the maximum values. The coefficient-of-variations among tasks in their requirements for CPU, memory, disk and network bandwidths are high at 1.42, 1.26, 2.24 and 2.05, respectively.

Demands for different resources are not correlated. Table 2 lists the correlation coefficients between pairs of resources. Even the highest correlation, between cores and memory, is only moderate. This is because the variety of specialized applications using these frameworks have focused requirements, e.g., compute- or IO-intensive. Hence overall, the tasks have *complementary* demands.

As a result of the diversity of resource demands and lack of correlation between them, utilization of different resources peaks at different times. Table 3 quantifies the fraction of times when the utilizations are "tight" (defined as, usage being higher than a certain fraction of capacity). Multiple resources become tight, albeit at different machines and times, thus calling for a packing solution that dynamically accounts for contended resources.

2.2.3 *Upper bounding potential gains*

How much can a scheduler that considers varying task requirements and resource usages help? Finding the optimal packing solution, especially considering dynamic job arrivals, is a daunting task (APX-hard). Instead, we solve a related but much simplified prob-

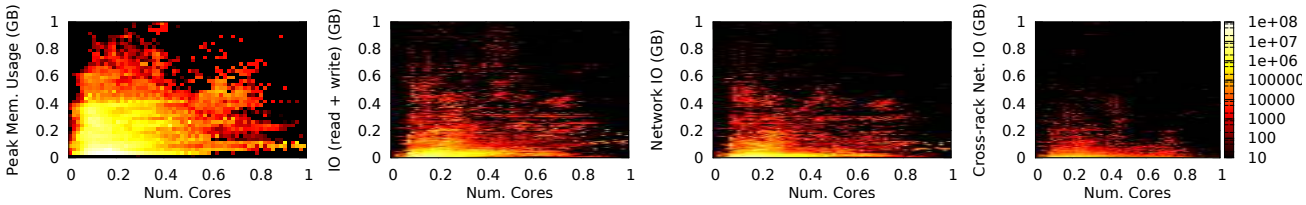


Figure 2: Heatmap of task requirements. Note the significant variation in the number of cores, memory, network and disk usage of tasks. The axes are normalized; the color of a point indicates the number of tasks, in logarithmic scale, having the corresponding resource demands.

	> 75% **used**	> 90% **used**	> 95% **used**
CPU	0.58	0.35	0.28
Memory	0.68	0.41	0.22
Disk in	0.11	0.02	0.003
Disk out	0.26	0.04	0.006
Network in	0.22	0.01	0.008
Network out	0.44	0.28	0.05

Table 3: Tightness of resources. Probability that a type of resource is used at above a certain fraction of its capacity in the Facebook cluster.

lem and hope that the optimal solution to the actual problem will not be better. In particular, this loose upper bound has these simplifying assumptions. First, it is offered an aggregated view of the cluster across each of its resources, i.e., one large bin per time rather than a bin per machine per time; hence, it does not have to worry about resource fragmentation at machine-level. Second, it assumes that tasks of a job (or phase) have the same resource requirements, thus reducing the search space; this assumption is mostly true (§4.1). Third, it avoids over-allocation explicitly by scheduling a task only when enough resources are available to meet its demands. We believe that gains for this simpler problem are an upper bound on the gains from optimal packing.

Our analysis shows that packing could reduce makespan (average job completion time) by 49% (46%) compared to slot-based fair scheduling and 38% (41%) compared to DRF. But, the gains are lopsided— 16% of the jobs slow down by 26%; the overall gains were calculated including the slowed-down jobs.

Motivated by the gains, we seek to design an online scheduler that packs tasks based on their requirements to improve cluster throughput and job completion times, while limiting the fallout due to any unfairness in the allocations.

3. Tetris SCHEDULER

In this section, we describe Tetris's scheduling heuristic, assuming complete knowledge of the resource requirements of tasks and resource availabilities at machines; we explain how to estimate these in §4. We begin with an analytical model that explains the computational complexity of our problem (§3.1). We then develop heuristics that match tasks to machines so as to reduce makespan (§3.2) and average job completion time (§3.3). Finally, we combine these heuristics with a large class of fairness algorithms (§3.4).

3.1 Analytical model

To connect the task scheduling problem with prior theoretical work, we cast it into a general optimization framework.

Notations: We consider demands of tasks along four resources: CPU, memory, disk, and network bandwidth (see Table 4). For every resource r, we denote (i) the capacity of that resource on machine i as c_i^r, and (ii) the demand of task j on resource r as d_j^r.

Next, we define variables that encode the task schedule and resource allocation. Note that allocation happens across machines (spatial) and over time (temporal). Let Y_{ij}^t be an indicator variable

Term	**Explanation**
$d_j^{\text{cpu}}, f_j^{\text{cpu}}$	CPU demand (i.e., cores) (d) and cpu cycles (f)
d_j^{mem}	Peak memory used by task
$d_j^{\text{diskR}}, d_j^{\text{diskW}}$, $f_{ij}^{\text{diskR}}, f_j^{\text{diskW}}$	Peak disk read/write bandwidth of the task (d), bytes to be read from machine i and output (f)
$d_j^{\text{netIn}}, d_j^{\text{netOut}}$	Peak network bandwidth in/out of a machine that the task can use (d)

Table 4: Resource demands of task j. Note that the demands for network resource depend on task placement.

that is 1 if task j is allocated to machine i at time t (time is discretized). Also, task j is allocated $X_{ij}^{r,t}$ units of resource type r on machine i, at time t. Let i_j^* denote the machine that task j is scheduled at and start_j and duration_j denote the corresponding aspects of task j.[6] We do not model preemption here for simplicity. Note that a task may need network and other resources at multiple machines in the cluster due to remote reads.

Constraints: First, the cumulative resource usage on a machine i at any given time cannot exceed its capacity:

$$\sum_j X_{ij}^{r,t} \le c_i^r, \forall i, t, r. \tag{1}$$

Second, tasks need no more than their peak requirements and no resources are allocated when tasks are inactive.

$$0 \le X_{ij}^{r,t} \le d_j^r, \forall r, i, j, t. \tag{2}$$

$$\forall_{i,r,t}\ X_{ij}^{r,t} = 0 \text{ if } t \notin [\text{start}_j, \text{start}_j + \text{duration}_j]. \tag{3}$$

Third, to avoid costs of preemption, schedulers may prefer to uninterruptedly allot resources until the task completes.

$$\sum_{t=\text{start}_j}^{\text{start}_j+\text{duration}_j} Y_{ij}^t = \begin{cases} \text{duration}_j & \text{machine } i = i_j^* \\ 0 & \text{other machines} \end{cases} \tag{4}$$

Fourth, the duration of a task depends on task placement and resource allocation:

$$\text{duration}_j = \max \left(\begin{array}{l} \frac{f_j^{\text{cpu}}}{\sum_t X_{i_j^* j}^{\text{cpu},t}}, \frac{f_j^{\text{diskW}}}{\sum_t X_{i_j^* j}^{\text{diskW},t}}, \\ \forall i \ \frac{f_{ij}^{diskR}}{\sum_t X_{ij}^{\text{diskR},t}}, \\ \frac{\sum_{i \ne i_j^*} f_{ij}^{diskR}}{\sum_t X_{i_j^* j}^{\text{netIn},t}}, \\ \forall i \ne i_j^*, \ \frac{f_{ij}^{diskR}}{\sum_t X_{ij}^{\text{netOut},t}} \end{array} \right) \tag{5}$$

[6] Though we use start_j and i_j^* in the constraints below for convenience, note that they are unknown apriori. Hence, all their occurrences need to be replaced with equivalent terms using the indicator variables Y_{ij}^t. For example $X_{i_j^* j}^{r,t} = \sum_i Y_{ij}^t X_{ij}^{r,t}$.

Term	Explanation
$c_i^{\text{cpu}}, c_i^{\text{mem}}$	Number of cores, Memory size
$c_i^{\text{diskR}}, c_i^{\text{diskW}}$	Disk read and write bandwidth
$c_i^{\text{netIn}}, c_i^{\text{netOut}}$	Network bandwidth in/out of machine

Table 5: Resources measured at machine i. Note that the availability of these resources changes as tasks are scheduled on the machine.

In each term, the numerator(f) is the total resource requirement (e.g., cpu cycles or bytes), while the denominator is the allocated resource rate (e.g., cores or bandwidth). From top to bottom, the terms correspond to cpu cycles, writing output to local-disk, reading from (multiple) disks containing the input, network bandwidth into the machine that runs the task and network bandwidth out of other machines that have task input. Note here that cores and memory are only allocated at the machine that the task is placed at but the disk and network bandwidths are needed at every machine that contains task input. We assumed here for simplicity that task output just goes to local disk and that the bandwidths are provisioned at each time so as to not bottleneck transfers, i.e.,

$$\forall t, j, i \neq i_j^* \;\; X_{ij}^{\text{netOut},t} \geq X_{ij}^{\text{diskR},t} \text{ and } X_{i_j^* j}^{\text{netIn},t} \geq \sum_{i \neq i^*} X_{ij}^{netOut,t}.$$

We also assumed that the task is allocated its peak memory size since unlike the above resources, tasks' runtime can be arbitrarily worse (due to thrashing) if it were allocated less than its peak required, i.e., $\forall t \;\; X_{i_j^* j}^{\text{mem},t} = d_j^{\text{mem}}$.

Objective function: Our setup lends itself to various objective functions of interest to distributed clusters including minimizing *makespan* [7] and *job completion time* [8]. Minimizing makespan is equivalent to maximizing packing efficiency. Further, we can pose fairness as an additional constraint to be meet at all times.[9]

Takeaways: The above analytical formulation is essentially a hardness result. Regardless of the objective, several of the constraints are non-linear. Resource malleability (eqn:2), task placement (eqn:5) and how task duration relates to the resources allocated at multiple machines (eqn:5) are some of the complicating factors. Fast solvers are only known for a few special cases with non-linear constraints (e.g., the quadratic assignment problem). Finding the optimal allocation, therefore, is computationally expensive. Note that these non-linearities remain *inspite* of ignoring the dependencies between tasks (e.g., the DAG) and the several simplifications used throughout (e.g., no task preemption). In fact, scheduling theory shows that even when placement considerations are eliminated, the problem of packing multi-dimensional balls to minimal number of bins is APX-Hard [27]. Worse, re-solving the problem whenever new jobs arrive requires online scheduling. For the online case, recent work also shows that no polynomial time approximation scheme is possible with competitive ratio of the number of dimensions unless NP=ZPP [9].

Given this computational complexity, unsurprisingly, cluster schedulers deployed in practice do not attempt to pack tasks. All known research proposals rely on heuristics. We discuss them in further detail later, but note here that we are unaware of any that considers multiple resources and simultaneously considers improving makespan, average job completion time and fairness considerations. Next, we develop heuristics for packing efficiency (§3.2), job completion time (§3.3) and incorporate fairness (§3.4).

[7] Makespan = $\max_{\text{job } J} \max_{\text{task } j \in J} \max_{\text{time } t} (Y_{ij}^t > 0)$

[8] Job J's finish time is $\max_{\text{task } j \in \text{ job } J} \max_{\text{time } t} (Y_{ij}^t > 0)$

[9] Dominant resource share of J at time t = $\max_{\text{resource } r} \frac{\sum_{i, j \in J} X_{ij}^{r,t}}{\sum_i c_i^r}$.

3.2 Packing Efficiency for Makespan

Packing tasks to machines is analogous to the multi-dimensional bin packing referred above. To develop an efficient heuristic, we draw an analogy with the solution in a one-dimensional space (both balls and bins). An effective heuristic proceeds by repeatedly matching the largest ball that fits in the current bin; when no more balls fit, a new bin is opened. Intuitively, this approach reduces the unused space in each bin (i.e., reduces fragmentation) and therefore, reduces the total number of bins used. In one-dimensional space, this heuristic requires no more than $(\frac{11}{9})$OPT+1 bins, where OPT is the optimal number of bins [25].

Alignment: We extend the above heuristic by defining *alignment* of a task relative to a machine across multiple dimensions. Similar to the one-dimensional case, the larger the alignment the lower the fragmentation. We considered many options for defining the alignment. Among them, the best packing efficiency was obtained using the dot product between task requirements and resource availabilities on machines.

Our allocation operates as follows. When resources on a machine become available, the scheduler first selects the set of tasks whose peak usage of each resource can be accommodated on that machine. For each task in this set, Tetris computes an *alignment score* to the machine. The alignment score is a weighted dot product between the vector of machine's available resources and the task's peak usage of resources. The task with the *highest* alignment score is scheduled and allocated its peak resource demands. This process is repeated recursively until the machine cannot accommodate any further tasks.

Picking tasks as above has these properties. First, only tasks' whose peak demands are satisfiable are considered; so over-allocation is impossible. Under-utilization of resources is possible, however, we discuss how to mitigate that using a resource tracker in §4.1. Second, the alignment score is largest for the task that uses the most resources on the machine along every resource type. Third, if a particular resource is abundant on a machine, then tasks that require that resource will have higher scores compared to tasks that use the same amount of resources overall. Together, this reduces fragmentation; loosely, bigger balls are placed first and machine resources that are currently abundant are used up before those that are currently scarce by choosing tasks appropriately.

When computing the dot product, Tetris normalizes the task requirements as well as available resources on the machine by the machine's overall capacity. This ensures that the numerical range of a machine's resource (e.g., 16 cores vs. 96GB of RAM) and tasks' demands (e.g., 4 cores vs. 1 Gbps network) do not affect the alignment score. All the resources are weighed equally.

Incorporating task placement: A subtle aspect in calculating the alignment score is to appropriately adjust the disk and network demands based on task placement. Recall that while CPU and memory allotment are primarily local, disk and network resources may be needed at multiple machines if data is read (or written) remotely. For remote reads, Tetris checks before placing a task on a machine that sufficient disk read and network-out bandwidth are available at each of the remote machines that contain task input and that sufficient network-in bandwidth is available at the local machine. Including these remote resources in the alignment score computation (dot product) is problematic. The vectors become as large as the number of machines in the cluster. Also, since the score is higher for tasks that use more resources overall, remote placement is preferred! Instead, Tetris computes the alignment score with only the local resources and imposes a remote penalty (of, say, 10%) to penalize use of remote resources. We run sensitivity analysis on the value of this remote penalty in §5.3.3. Together, this ensures that over-

allocation is still impossible, fragmentation is reduced and using local resources is preferred. Consequently, the network and remote disks are free for tasks that compulsively need them.

3.3 Average Completion Time

Achieving packing efficiency does not necessarily improve job completion time. Consider two jobs running on a two machine cluster. Machines have 2 cores and 4GB of memory. The first job has six tasks each requiring ⟨2 cores, 3 GB⟩ while the second job has two tasks requiring ⟨1 cores, 2 GB⟩. Tasks of the first job have higher alignment score. Hence, the above packing efficiency heuristic schedules all tasks of the first job before the second job's tasks. Assuming all the tasks have the same duration, it is easy to see that swapping the order lowers average job completion time.

A useful starting point to reduce average job completion time is the shortest-remaining-time-first (SRTF) algorithm [16]. SRTF schedules jobs in ascending order of their remaining time. While one can extend SRTF to consider multiple resource requirements of tasks, its greedy scheduling approach nonetheless leads to resource fragmentation [20]. The consequent loss in packing efficiency delays the scheduling of other waiting jobs, thus inflating their durations. Therefore, the challenge in improving job completion time is to *balance prioritization of jobs that have less remaining work against loss in packing efficiency*.

3.3.1 Shortest Remaining Time

We first extend the SRTF scheduling algorithm to incorporate multiple resources. Consider scheduling jobs in increasing order of their number of remaining tasks. The fewer tasks remaining in a job, the lower is its remaining duration. However, tasks can have different durations. Worse, tasks can have varying requirements for different resources. Scheduling a job with high resource requirements, even if it has the smallest duration, incurs the opportunity cost of not scheduling other jobs simultaneously. Intuitively, favoring jobs that have the least remaining work improves average completion time the most with the least effort.

Scoring job's remaining work: We calculate the total resource requirements of all remaining tasks of the job across all dimensions. To be immune to variations in numerical ranges of the different resources, we normalize the various resources as before. The resource consumption of a task is the sum across all the (normalized) resource dimensions multiplied by the task duration. We estimate task duration based on earlier runs of the same or similar jobs as well as the completed tasks of the same phase. The score for a job is the total resource consumption of its remaining tasks.

Scheduling jobs with lower scores first reduces average completion time the most. The score prefers jobs with low duration mimicing the SRTF heuristic. In addition, favoring jobs that need less resources has smaller opportunity cost since more resources are available for other jobs.

3.3.2 Combining with Packing Efficiency

Recall that we need to combine the above score with the packing efficiency heuristic since using only the above score can fragment resources which delays jobs and inflates cluster makespan.

We combine the two metrics —remaining resource usage and alignment score—using a weighted sum. Denoting the alignment score by a and the remaining resource usage as p, we pick the task with the highest value of $(a + \varepsilon \cdot p)$. To ensure that the expression is not dominated by either, we recommend $\varepsilon = (\bar{a}/\bar{p})$, where $\bar{a}$ and $\bar{p}$ denote the average scores for alignment and remaining work. Note that a is per-task, whereas p is per-job. While the best choice of ε depends on the workload, our evaluation shows that the gains from packing efficiency are only moderately sensitive to ε since most jobs have pending tasks with similar resource profiles. Improving average job completion time, however, requires non-zero values of ε though the gains stabilize quickly. We will show in §5.3 that the above combination significantly improves over alternatives.

3.4 Incorporating fairness

As described so far, Tetris matches tasks to machines to improve cluster efficiency and job completion time without considering fairness. Fairness is a useful tool in shared clusters. Conversations with cluster operators indicate that proportional allocations improve performance predictability. However, it is not an absolute requirement and deviations in fairness for better performance are acceptable. Other properties such as strategy-proofness, envy freedom and sharing incentive are less important in private clusters since the organization can mandate sharing and audit jobs post facto. Here, we describe how Tetris balances performance with fair allocations.

A common property to many fairness based schedulers is to keep track of the allocations of running jobs and offer the next available resource to the job that is farthest from its fair share. Slot based fairness [3, 4] has this property: the next slot is allocated to the job that occupies the fewest slots relative to its fair share. Likewise, DRF [12] offers resources to the job whose dominant resource's allocation is furthest from its fair share. Similar to DRR scheduling, long-running or resource-hungry tasks (alt: large packets) cause short-term unfairness during such allocation. However, bounded task sizes (as in our clusters) limits the unfairness and, in practice, the above method offers fairness at meaningful timescales.

Tetris uses a *fairness knob* to trade-off fairness for performance. The knob f takes values in the range of $[0, 1)$ and intuitively quantifies the extent to which Tetris adheres to fair allocations. When resources become available, Tetris sorts the jobs (set J) in decreasing order of how far they are from their fair share. Note that most fair schedulers already maintain such a list. It then looks for the best task (as before) among the runnable tasks belonging to the first $\lceil (1 - f)\ J \rceil$ jobs in the sorted list. Setting $f = 0$ results in the most efficient scheduling choice, whereas $f \to 1$ yields perfect fairness. In this way, Tetris can incorporate most policies for fairness.

The trade-off between fairness and performance is rarely smooth. However, in scheduling tasks in data-parallel clusters, we find that the above knob lets Tetris achieve schedules that are both reasonably fair and reasonably efficient. For knob values in the range [0.25,0.5], Tetris's makespan is within 10% of the most-efficient (and most-unfair) schedule while delaying jobs by no more than 10% compared to the fair schedulers. This is because though the fairness constraint restricts which jobs (or queues) the available resources can be offered to, there are many tasks pending in a job or queue and from among these many choices it is still possible to pick a task that aligns well with the resources on the machine to be allocated, leading to nearly the same packing efficiency.

3.5 Discussion

DAG Awareness: The last few tasks before a barrier (e.g., sometimes, no reduce task can start before all maps finish) present a low-cost high-gain opportunity. For the purpose of this discussion, the end of a job can also be considered a barrier. Delay in scheduling these tasks directly impacts job completion time. However, scheduling just these last few tasks preferentially has little impact on other jobs since it is taking away only a small amount of resources. Tetris uses knowledge of the job's DAG to preferentially schedule the last few tasks before a barrier. Given a barrier knob value $b \in [0, 1)$, whenever resources are available Tetris preferentially offers them to tasks that remain after b fraction of tasks in the stage preceding a

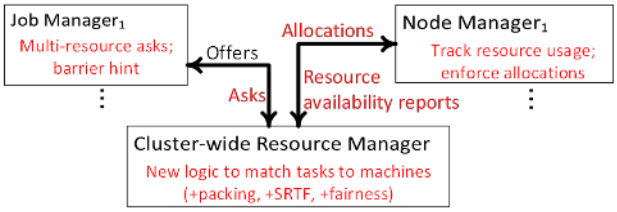


Figure 3: Typical Architecture of a Data-Parallel Cluster Scheduler and the changes to add Tetris (shown in red).

barrier have finished. Among these tasks, Tetris still ensures that sufficient resources are available and chooses based on alignment score. Results in §5.3.3 show that values of $b \in [.85, .95]$ improve both job completion time and makespan.

Starvation Prevention: Initially, we were concerned that the above solution can *starve* tasks that have large resource demands. Since the scheduler assigns resources on a machine as soon as they are available, it could take a long time to accommodate large tasks. Note that DAG awareness kicks in only for the last few tasks of a job, and hence may not help. However, in practice, we do not encounter such starvation due to two reasons. First, recall from Figure 2 that most tasks have demands within one order of magnitude of one another. As a result, it is likely that the machine resources that become free when one or a small number of tasks finish suffice to fit even the larger tasks. Second, machines report resource availability to the scheduler periodically (§4.4). The intended goal was to keep the scheduler load low. But, doing so also causes the scheduler to learn about all the resources freed up by the tasks that finish in the preceding period together. A more principled solution that reserves machine resources for *starved* tasks is left to future work.

Future Demands: While our solution is online, we point out that (imperfect) knowledge of the future is already available. Since each job manager knows its DAG, it knows the future tasks. The scheduler can also estimate when future machine resources become available; each job manager can estimate when an assigned task will finish similar to how they estimate tasks' resource demands. Using such future knowledge, the scheduler can better mimic the offline solution. We leave to future work how to extend the scheduler to use such future knowledge.

4. SYSTEM DESIGN

In this section, we describe how Tetris estimates task requirements and resource availability at machines, and how to add Tetris to cluster schedulers.

4.1 Estimating task demands and available resources

Recall from Tables 4 and 5 that Tetris considers the following resources for scheduling: CPU, memory, disk and the network. Since clusters today have small over-subscription factors in the core [13], a simplification is to only consider the network bandwidth on the last hop, i.e., the link between the machine and the ToR switch.

Per resource, Tetris has to estimate their dynamic availability at machines. To do so, Tetris employs a resource tracker process that runs at every node in the cluster; the tracker observes aggregate resource usages from operating system counters and periodically reports to a cluster-wide resource manager that handles scheduling.

Also, per-resource Tetris has to estimate tasks' peak demands (e.g., rate of reading in Gbps). Here, Tetris uses these ideas. First, the size and location of inputs of tasks are known before their execution. Second, *recurring jobs* are fairly common in clusters [7]; analytics jobs repeat hourly (or daily) to do the same computation on newly arriving data. For such jobs, Tetris uses task statistics measured in prior runs of the job. Third, since tasks in a phase perform the same computation on different partitions of data, their resource use is statistically similar. Hence, Tetris estimates the demands of later tasks in a phase using the measured statistics from the first few tasks. In our traces we find that the coefficient-of-variation in resource use among tasks in a phase is 0.13, 0.022, 0.25 and 0.41 for CPU, memory, disk and network bandwidths respectively at median. Many phases with large CoV have only a small number of tasks. Other phases have greater variation on the low-end, e.g., tasks placed locally do not use network bandwidth. Fourth, when none of the above methods are available, such as for the first few tasks in a phase, Tetris over-estimates their task demands. Over-estimation is better than under-estimation which needlessly slows down tasks. However, resources can remain idle due to such over-estimates. Tetris relies on the resource tracker to report unused resources and allocates them to new tasks. In its reports, the tracker provides allowance for newly assigned tasks to "ramp up" their usages. It does so by increasing the observed usage by a small amount per task; the amount decreases over the task's lifetime and goes to zero after a threshold (we use 30*s*). Finally, there has been recent work on static program analysis to estimate resource demands [14]. The techniques are not yet applicable to arbitrary programs but represent a potential solution for the future.

4.2 Enforcing allocations

Due to arbitrariness in the user code, tasks' actual resource use may not conform to their allocations. For e.g., a TCP flow can ramp up to use all the available network bandwidth. To guard against this, Tetris explicitly *enforces* allocations.[10] For disk and network usage, Tetris intercepts all calls to read or write from the filesystem, including remote reads. Each call is routed to a corresponding token bucket that allows the call to go through if enough tokens remain, or queues it otherwise. Tokens arrive at the allocated rate, the size of the bucket determines the maximum burst size, and each call deducts a corresponding number of tokens when it goes through.

4.3 Ingest, evac., and other cluster activity

The resource tracker also helps the scheduler to be aware of other activities in the cluster. In addition to executing jobs, data ingestion and maintenance operations can use up a lot of cluster resources. Ingestion refers to storing incoming data for later analytics; some clusters report volumes of up to 10TB per hour [2, 6]. Prior to maintenance operations on machines, their data is evacuated and re-replicated to ensure that availability is unchanged. By using the tracker's reports, the scheduler can avoid contention between its tasks and these activities.

4.4 Adding Tetris to cluster schedulers

Figure 3 depicts the core pieces of typical cluster schedulers today and shows in red the new functionality to be added to each piece for Tetris. For scale, performance and reliability, today's data-parallel cluster schedulers, as well as earlier grid computing schedulers [23], breakdown cluster scheduling into three parts. A node manager runs on every node in the cluster, responsible for running tasks and reporting node liveness. Per job, a job manager process runs on some node in the cluster and holds job context such as task progress and dependencies (e.g., DAG). A cluster-wide resource arbiter receives *asks* from various job managers for their pending tasks; an ask

[10] For memory and CPU cores, Tetris relies on existing support in Yarn; it uses OS mechanisms for the latter and explicitly tracks the total memory used by the task and its child processes for the former.

encodes preferences for data locality etc., and in turn *offers* guidance as to where the tasks can run.

We incorporated Tetris into the Yarn framework (Hadoop 2.4). [11] To do so, we made these modifications:

1. Matching tasks to machines, at the cluster-wide resource manager, now implements the ideas in §3 (packing, multi-resource SRTF, fairness knob, barrier hints).
2. The job manager estimates task demands from historical jobs and from earlier tasks in the phase (§4.1); asks from the job manager report task demands across multiple resource types and also identify the last few tasks before a barrier (§3.5).
3. The node manager now observes dynamic resource availability across various resources and enforces task allocations.

These changes do not impact the underlying complexity and hence the scheduler's scalability with two exceptions. First, since task placement changes its demands for disk and network, e.g., losing data locality adds need for network bandwidth, how can the job manager specify demands in its ask before the task is placed? If the ask were to contain task demands for each possible placement, it would be too large. Tetris keeps the asks succinct by observing that given the locations and sizes of a task's inputs, its resource demands can be inferred for any potential placement. Second, the new logic to match tasks to machines is more complex; we report overhead measured using our prototype in §5.2.2.

5. EVALUATION

We evaluate Tetris using our prototype implementation on a 250 machine cluster. To understand performance at larger scale, we do trace-driven simulations using traces from production clusters.

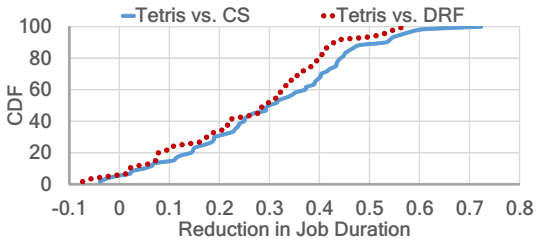
5.1 Setup

Cluster: On the larger, 250 server, cluster each machine has 32 cores, 96GB of memory, 8 1TB 7200RPM disk drives, a 10Gbps NIC and runs Linux 2.6.32. The machines are 40 to a rack connected in a folded CLOS with 1.25X over-subscription between racks. We also report experiments on a smaller, 7 server, cluster. Here each machine has 6 cores, 12GB of memory, 3 1TB disks and a 1Gbps NIC.

Baselines: We compare Tetris to state-of-the-art scheduling algorithms implemented in Hadoop 2.4. The Fair and Capacity Schedulers [3] are deployed in production clusters at Facebook and Yahoo! respectively. Conceptually, both schedulers allocate slots (defined on memory) to jobs based on fairness; we use a slot size of 2GB similar to the Facebook cluster. Dominant Resource Fairness (DRF) [12] is a recent multi-resource fairness algorithm that is available with YARN. DRF considers memory and CPU requirements. To the best of our knowledge, none of the existing schedulers consider disk or network bandwidths in their allocations. Further, the schedulers try to place map tasks on local slots [29] and reduce tasks on any available slot. Finally, we also compare Tetris's gains to the simple upper bound described in §2.2.

Trace-driven Simulator: To evaluate Tetris at a larger scale and under many more parameter choices, we built a trace-driven simulator that replays logs from Facebook's production clusters. The simulator mimics these aspects from the original trace: job arrival times, task resource requirements, the input and output sizes of tasks, the location of inputs in HDFS and the probabilities of failures. Further, we use the same machine profile as the original cluster: 12 cores, 32GB memory, 4 disks operating at 50MBps each for read/write and a 1Gbps NIC.

[11] In Yarn's terminology, the Job Manager in Fig. 3 is referred to as the application master (AM), the Cluster-wide resource manager is called the RM and the node manager is NM.



(a) CDF of change in Job Completion Time

	Avg.	Stdev.
Tetris **vs. CS**	29%	2.94%
Tetris **vs. DRF**	27.5%	3.50%

(b) Makespan Reduction

Figure 4: Comparing Tetris vs. baselines. The results are from running a workload of over 100 jobs on a 250 server cluster running Yarn 2.4. Each experiment run takes > 10 hours; repeat five times.

Workload: To test our prototype, we constructed a workload suite of over 100 jobs by picking uniformly at random from the following choices. Job size (number of tasks) and the selectivity of map and reduce tasks are chosen from one of four choices: large & highly-selective, medium & inflating, medium & selective and, small & selective. The ratios of input to output are: 1:2 for inflating, 1:0.5 for selective and 1:0.2 for highly selective. The sizes are: couple 1000 tasks for large, 1000 for medium, 100s for small. A map- or reduce- stage could either have tasks of high-mem (8GB) or low-mem (2GB). Similarly the stage could either have tasks with high-cpu or low-cpu; the former indicates that tasks do substantial computation per data read or written and hence have low peak I/O demands. Job arrival time is uniformly picked at random between [0:3600]s. Each experiment run takes about five hours and we repeat five times.

Metrics: Our figures of merit are improvements in job completion time, makespan and the extent of unfairness. For the first two metrics, we compute the percentage improvement (or reduction) as $100 \times \frac{baseline - tetris}{baseline}$. 25% improvement means Tetris is 1.33X better. We report the changes at the average as well as the distribution across jobs. To quantify unfairness, we report the fraction of jobs that slow down (and by how much) compared to a fair allocation.

Highlights: The highlights of our evaluation are as follows.

1. Tetris improves cluster makespan and average job completion time by about 30% in our deployment and by up to 40% when replaying Facebook traces in simulation. We estimate that these gains are 80% of the simple upper bound §2.2.3. The gains accrue from avoiding fragmentation and over-allocation and from exploiting complementary demands for better packing.
2. Tetris offers the above efficiency gains with only a slight loss in fairness. These gains use fairness knob $f = 0.25$ and fewer than 6% of jobs are delayed by no more than 10% compared to a fair allocation. Setting $f = 0$, the most efficient and unfair schedule, improves makespan by up to 50% but the ensuing unfairness delays more jobs. Surprisingly $f = 1$, the most fair schedule, also offers sizable improvements in efficiency.
3. We compare against several alternatives and perform a detailed sensitivity analysis on parameters.

5.2 Deployment Results

Figure 4 compares the performance of Tetris with baseline schemes. Each experiment runs our workload (described above) on the cluster. We see that Tetris improves job completion time, at median, by 30%. The top fifth of jobs improve by over 40%. Gains are

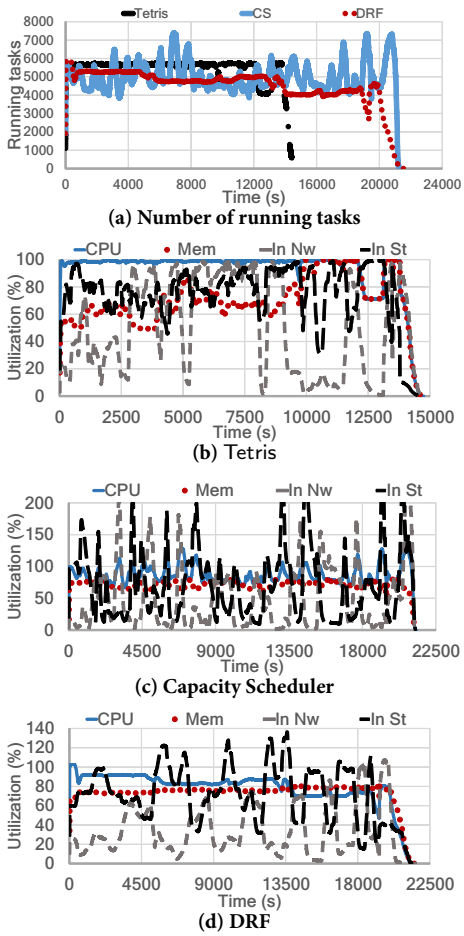


Figure 5: Number of running tasks and total resources used during an example experiment run.

slightly larger compared to the capacity scheduler (CS) than with DRF. Makespan also improves by about 30%.

To understand in a bit more detail, Figure 5 shows the details for one example run. A couple of points stand out. First, we see from Fig. 5a that Tetris keeps consistently high numbers of running tasks in the cluster. DRF, at all times, has scheduled fewer tasks. Capacity scheduler (CS) has slightly higher throughput but is less consistent. Partly, this is because neither DRF nor CS explicitly avoid resource fragmentation. Further, DRF is not pareto-efficient when allocating resources spread across multiple machines; it uses a simpler one-machine model [12]. More importantly, pareto efficient fair allocations do not lead to the best performance, as we saw in the example; partly, this is due to the DAG structure of jobs and tasks having different resource profiles. Second, Fig. 5b shows the (estimated) usage of resources across the cluster. We see that, with Tetris, the cluster is bottlenecked on different resources at different times. Comparatively, we see in Fig. 5c that the capacity scheduler is unable to fully use even the resources that it considers explicitly for scheduling (memory and CPU). This is due to fragmentation. Worse, it over-allocates the other resources (network and disk cross 100% at several times). DRF is slightly better but qualitatively similar. Third, we observe that due to efficient packing, not only is the bottleneck resource fully utilized at all times but also the utilization of other resources increases. For example, the network and disk usages are markedly higher with Tetris compared to the baselines. We summarize that avoiding fragmentation and over-allocation by

	CPU	Memory	Disk	Network
Tetris	.90, .82, –	.47, .31, –	.41, .24, –	.55, .30, –
CS	.87, .33, .18	.38, –, –	.29, .19, .13	.42, .34, .28
DRF	.53, .13, –	.56, –, –	.12, .07, .05	.40, .27, .20

Table 6: Probability that a machine is highly using a type of resource at above a certain fraction (> 75%, > 90%, > 100%) of its capacity.

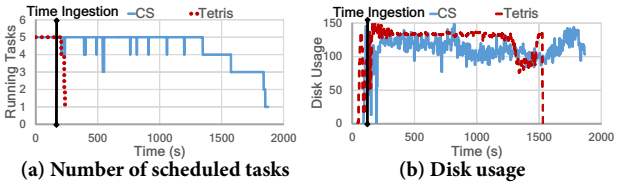


Figure 6: On a machine overloaded by data ingestion, Tetris stops scheduling further tasks. CS does not react; the resulting contention lowers disk throughput slowing down both tasks and ingestion.

Time to process	CS		Tetris	
	15K	(45K) tasks	15K	(45K) tasks
NM heartbeat	.06ms	(.12ms)	.06ms	(.07ms)
AM heartbeat	.2ms	(.2ms)	.15ms	(.15ms)

Table 7: Overheads: Average time to process heartbeats from the NM and the AM with and without Tetris; vary tasks pending; 250 servers.

explicitly packing on all the relevant resources improves efficiency. The gains accrue from (a) increase in the number of simultaneously running tasks, (b) reduced task duration (by not over-allocating) and (c) improved usage of all cluster resources.

Table 6 depicts aspects of machine-level resource usage during an example run. It measures the likelihood of a machine to have a certain usage for each resource. We confirm that Tetris is able to use more of all resources (by avoiding fragmentation). Baseline schemes under-use due to fragmentation and occasionally over-allocate the disk and network.

5.2.1 *Micro-benchmark: resource tracker*

We use micro-benchmarks to highlight aspects that are not as easy to see from the large workload runs above.

To evaluate how Tetris adapts to cluster activity, we mimic ingestion on a machine in our small cluster. Fig. 6 shows the number of running tasks on that machine as well as the disk load of the machine; ingestion begins at t = 125s. We see that the resource tracker used by Tetris observes the rising disk usage and schedules no more tasks on that machine. Tasks already scheduled complete at t = 250s. In contrast, the capacity scheduler proceeds unaware; the resulting contention lowers disk throughput leading to both straggler tasks and delays in ingestion.

5.2.2 *Overheads and Scalability*

Recall from our prototype description (§4.4) that Tetris's logic to match tasks to machines is more complex than that in Yarn. To estimate overheads, we ran experiments with different cluster sizes and different numbers of jobs and pending tasks. We find that the additional network overhead, due to expanding the size of the *asks* is negligible. Further, the memory usage at the cluster-wide resource manager increases very slightly; the timeline graphs (not shown) are dominated by when garbage collection happens. For the computational overhead, we observe the time taken by the cluster-wide resource manager (RM) to process a *heartbeat* from the node manager (NM) and from the job manager (AM). Yarn's RM does the actual allocation during the NM heartbeat; AM asks are cumulative, so at an AM heartbeat, the RM simply updates the cumulative ask and responds with any tasks in the (past) asks that have been

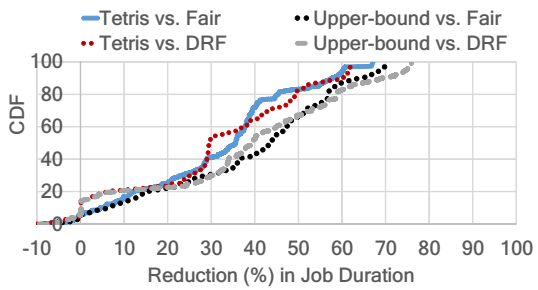


Figure 7: CDF of improvement in job completion time. The top 20% of jobs improve by over 50%.

matched (during the NM heartbeats) since the AM's previous ask. We see from Table 7 that Tetris takes about the same time for an AM heartbeat, but is slightly faster on the NM heartbeat esp: when more tasks are pending. This is because default Yarn RM does some extra work per heartbeat that is not needed given Tetris. Despite the fact that our prototype is not optimized, we estimate that an RM using Tetris scales similarly to the default Yarn. Recall that scalability was a key reason behind our choice to avoid more complex solutions based on flow-networks and integer linear programming [18].

5.3 Trace-driven Simulations

Here, we mimic scheduling in a several thousand server facebook cluster. Unless otherwise specified, results use Tetris's default parameter values (fairness knob $f = 0.25$, barrier knob $b = 0.9$, $\varepsilon = \frac{\bar{a}}{\bar{p}}$). We measure sensitivity to parameters in §5.3.2 and §5.3.3.

5.3.1 *Efficiency*

Job Completion Time: Compared to the fair scheduler and DRF, Tetris speeds up jobs by 40% and 35% on average. Figure 7 shows the CDF of the improvement in job completion time over jobs in the trace. We see that the top fifth of jobs speed-up by over 50%. The gains at the median are over 30%. Tetris's gains are slightly more relative to the fair-scheduler in Hadoop than in DRF but not markedly so. Further, Tetris's average gains are 81% of those from the simple upper-bound scheduler. Recall that the simple upper-bound ensures minimal fragmentation and avoids over-allocation but is not a true upper bound; the true upper bound is intractable to compute. A few jobs slow down– fewer than 6% of jobs by less than 10% each. These slow downs are because Tetris trades off fairness for efficiency.

Where do the gains come from? We find that task durations improve by about 25%. Since resource fragmentation has no impact on task duration, we believe task durations improve because of avoiding over-allocation. Reduce tasks improve more than the map tasks. Recall that the baseline schedulers do not consider disk- or network-bandwidths when scheduling tasks and can over-allocate these resources causing tasks to contend needlessly for disk and network while holding on to the other resources. If Tetris were modified to consider only CPU and memory (i.e., it also over-allocates disk and network bandwidths), the average gains drop from 40% to 15% for Tetris vs. Fair and 35% to 13% for Tetris vs. DRF. Hence, nearly two-thirds of the gains are due to avoiding over-allocation, and one-third due to avoiding fragmentation.

Further, we find that while jobs of all sizes see speed-up in completion time, large jobs ($\geq$ 1000 tasks) see a greater benefit; over 50% (not shown). This is likely because avoiding fragmentation allows Tetris to schedule more of their tasks early. Gains for small jobs ($\leq$ 25 tasks), however, are still significant at 35% due largely to our use of SRTF heuristic. Using *only* the SRTF heuristic lowers the improvement in average completion time to 27% and 26% over the fair scheduler and DRF, respectively. The corresponding numbers from using *only* the packing efficiency heuristic are 31% and 28%. We conclude that Tetris's combination of the two heuristics is better than using either individually.

Alignment Heuristic	Avg. Gain in Job Completion Time (%)	Gain in Makespan (%)
Cosine Similarity	40%	41%
L2-Norm-Diff	30%	40%
L2-Norm-Ratio	33%	36%
FFD-Prod	25%	24%
FFD-Sum	26%	31%

Table 8: Cosine similarity outperforms the alternatives for calculating the alignment score.

Makespan: To analyze the change in makespan, we assume that all jobs arrived at the beginning of the trace and measure the time until the last job finishes. Tetris reduces makespan by 33% and 24% compared to the fair scheduler and DRF. These gains are less than the simple upper bound scheduler's makespan reductions of 49% and 38%, respectively. The main reason is that Tetris's preference for jobs with less remaining work loses some of the makespan gains in favor of reducing average job completion time. In fact, just using the packing heuristic ($\varepsilon = 0$) improves makespan by a bit more, 41% and 29% respectively. We believe $\varepsilon = \frac{\bar{a}}{\bar{p}}$ to be a reasonable compromise; we evaluate sensitivity on ε in §5.3.3.

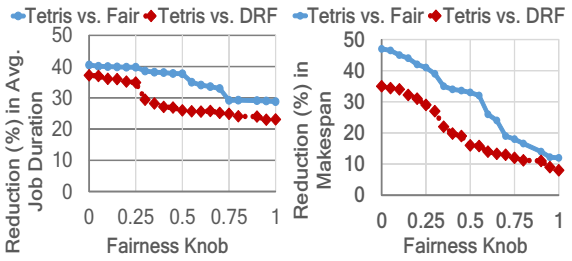
Alternative Packing Heuristics: Tetris's use of cosine similarity to calculate the alignment score performed the best among several alternatives proposed in literature [20, 25]. For a task with demands d to be scheduled on a machine with available resources a, cosine similarity is $a \cdot d$ where both the a and d vectors are normalized to the machine's capacity. To compare, L2-Norm-Diff is $\sum_i (d_i - a_i)^2$, L2-Norm-Ratio is $\sum_i \left(\frac{d_i}{a_i}\right)^2$, FFD-Prod is $\prod_i d_i$ and FFD-Sum is $\sum_i d_i$ [20]. Table 8 has our findings. Gains in both completion time as well as makespan are clearly better with cosine similarity; L2-Norm-Diff does well on makespan but lags behind in speeding up jobs. We conclude by noting that better heuristics may exist.

5.3.2 *Fairness vs. Efficiency*

Recall from §3.4 that Tetris offers a trade-off between fairness ($f \to 1$) and efficiency ($f = 0$). Figure 8 shows job completion and makespan for different values of this knob f.

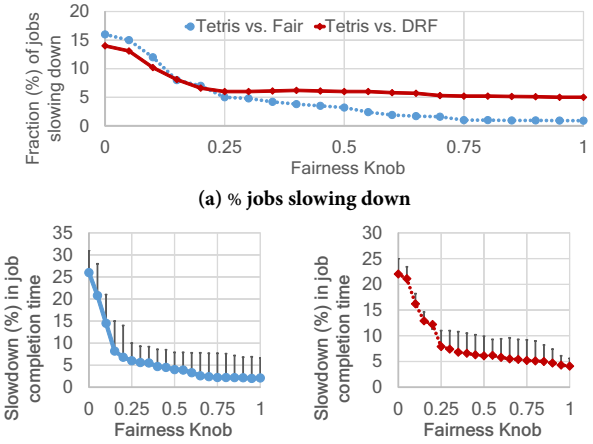
Couple points stand out. First, even with $f \to 1$, wherein available resources are always allocated to the job that is furthest from fair share, Tetris offers sizable gains. Makespan improves by 10% and average job duration improves by 23%. This is because even when restricted to the subset of allocations that are fair (e.g., which job(s) to give resources to) there are many choices (e.g., which of the pending tasks in those job(s) to schedule). By picking a suitable task from the constrained set of jobs, Tetris improves efficiency while being fair. Second, knob values around 0.25 achieve most of the gains. Job completion time gains plateau beyond $f = 0.5$. While makespan continuously improves by reducing f, $f = 0.25$ is less than 10% away from the best possible gains.

Finally, we report the impact of unfairness on jobs. Whereas most jobs finish faster under Tetris, some jobs slow down. We can measure the impact of Tetris's unfairness through the prevalence and magnitude of job slow-down. Figure 9 shows the fraction of jobs that slow-down and the average (error-bars=worst) per-job slowdown due to unfairness compared to the fair scheduler and DRF. Whereas $f = 0$ can slowdown up to 15% of jobs, $f = 0.25$ has nearly the same impact as choosing $f \to 1$; about 5% of jobs slow down by



(a) Job Duration (b) Makespan

Figure 8: Impact of varying the fairness knob. We see that setting a knob value around 0.75 offers nearly the best possible efficiency.



(a) % jobs slowing down

(b) Baseline: Fair (c) Baseline: DRF

Figure 9: Job Slowdown. For fairness knob values $\in [0.25, 0.5]$, only a few jobs slow down and only by a small amount.

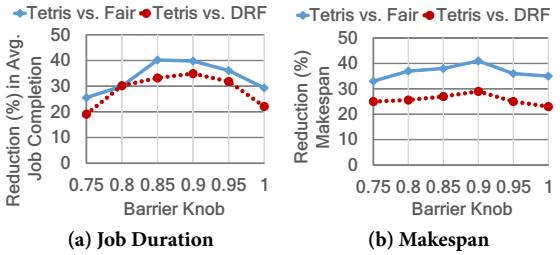
about 5% each. Notice that even $f \to 1$ slows-down some jobs; some of this is statistical noise; some of it is because even with $f \to 1$, Tetris picks tasks from the job in an order that is appropriate for packing which may not be the best for job's runtime (for example if a long-running task is scheduled later).

Relative Integral Unfairness: We now employ a stricter metric to quantify unfairness: if $a(t)$ and $f(t)$ are the actual allocation received by a job at time t and its purported fair allocation, we measure $\left(\int_t \frac{a(t)-f(t)}{f(t)} \mathrm{d}t\right)$ where the integral runs over the job's runtime. We call this the relative integral unfairness. Jobs that have a value below zero have received worse service due to Tetris than they would have in a fair scheme; value above zero implies job was treated better by Tetris. We find only a few jobs have negative values (about 5%) and the average negative value is small (about 5%). This hints that Tetris's violations of fair allocation are *transient*; when measured across the lifetime of a job the net violation (and hence, slowdown) are small.

5.3.3 *Sensitivity Analysis*

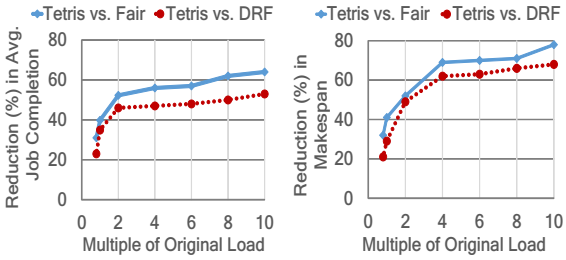
Here, we evaluate Tetris under different conditions.

Barrier Knob: Recall from §3.5 that Tetris preferentially schedules the last few tasks in a stage preceding a barrier after a b fraction of tasks finish. Figure 10 shows the impact of changing b. When $b = 1$, no tasks are preferentially treated. Lower the value of b, the larger the fraction of tasks that receive preference. Doing so will speed up the corresponding job but can delay other jobs since it takes resources away from them. Figure 10 shows that $b < 0.75$ is worse than not using the barrier promotion at all (i.e., $b = 1$). However, setting b around .9 is net positive. For e.g., Tetris vs. DRF, makespan reduction increases from 22% to 30%. Across different subsets of jobs



(a) Job Duration (b) Makespan

Figure 10: Barrier threshold value of 0.9 balances stagnation-avoidance with the loss from not picking the best task for packing.



(a) Job Duration (b) Makespan

Figure 11: Tetris's gains increase as the cluster load increases.

and traces from different days, we found $b \in [.85, .95]$ to be a good choice for the Facebook cluster.

Remote Penalty: Tetris uses a penalty in its alignment score (of 10%) when remote resources are used by a task. The penalty is a simple way to prefer local placement whenever possible. Our analysis shows that both completion time and makespan change little when the penalty value is between 6% and 15%. On either side of this range, gains drop moderately due to over-using remote resources or being too conservative and letting them lie fallow.

Weighting Alignment vs. SRTF: Recall that we combine alignment score (a) and remaining work (p) using a weighted sum, $(a + \varepsilon \cdot p)$ (§3.3.2). Our default choice of ε is $\bar{a}/\bar{p}$ where $\bar{x}$ is the average value of x. Here we evaluate the sensitivity of ε. Let $m = \varepsilon\bar{p}/\bar{a}$. While gains in completion time drop by about 10% for $m = 0$, they do not improve beyond $m = 1.5$. Gains in makespan are higher by 8% for $m = .5$ and do not drop much beyond $m = 1.5$. Thus, using a value of m close to 1 is appropriate.

Cluster Load: Utilization of a cluster impacts the possible gains from better packing/ scheduling. For example, if the load is one task at a time, packing is not required. The cluster whose trace we replay is only moderately loaded. Figure 11 examines the performance of Tetris under different cluster load. We vary load by changing the number of servers; half as many servers leads to twice the load. As expected, the figure shows that the gains due to Tetris improve with cluster load. At 4× the current load, Tetris improves makespan by well over 60% and average completion time by over 50%. Several studies show that faster job completion cause more jobs to be submitted to the cluster [6, 21]. Hence, techniques such as Tetris could be more useful in the future.

6. RELATED WORK

Tetris's contributions are, (a) multi-resource packing of tasks to machines, and (b) simultaneous support for complementary objectives of fairness, minimizing job completion time and minimizing cluster makespan. We contrast these contributions with some existing work. We did relate to well-known algorithmic results on bin packing earlier (§3.1); we mention a few more here.

Cluster Schedulers: Early cluster schedulers [3, 18, 4] focused primarily on maximizing data locality, scale and fairness. Dominant Resource Fairness [12] allocates multiple resources in a manner that is pareto-efficient, strategy-proof, envy-free and incentivizes sharing. Available implementations of DRF and the earlier schedulers only consider CPU and memory. We show that the focus on fairness detracts considerably from performance; that ignoring disk and network leads to harmful over-allocation; and posit multi-dimensional packing to be a key aspect of cluster scheduling.

Network Optimizations: Allocating IO is not a simple extension to multi-resource schedulers. Unlike CPU and memory which are purely local resources, the IO demands of a task depend on its placement relative to its input locations. Some prior work focuses on fair network allocation even when the applications are uncooperative [22], as well as better ways to schedule network transfers [10] and to steer around network hotspots [11]. Here, Tetris shows how to seamlessly choose between local and remote task placements and to allocate network and disk bandwidths to tasks so as to improve job and cluster performance.

Fairness vs. Cluster Efficiency vs. Job completion time: We are hardly the first to explore this trade-off. There is notable performance analysis on the impact of preferring short jobs [28] and on the unfairness of schedulers [26]. While less analytical, our key contribution here is a practical system that reaches a good operating point in this trade-off space. We also notice how in data-parallel clusters the tradeoff is smoother since one can be fair to jobs and still improve packing by scheduling an appropriate task.

Bin Packing: We build upon algorithmic work on multi-dimensional bin packing. Their goal translates directly to minimizing cluster makespan. We rely on them to understand the computational complexity of our problem [9, 27] and use their heuristics [20, 25] as a starting point. We extend to handle practical concerns (placement, resource malleability) and add support for complementary objectives (minimize job completion time; fairness).

Virtual Machine Packing: A related problem involves consolidating VMs, with multi-dimensional resource requirements, on to the fewest number of servers [19]. The best work there focuses on different challenges and is not applicable to task packing. For e.g., they balance load across servers, ensure VM availability inspite of failures, allow for quick software and hardware updates etc. They have no corresponding entity to a job and hence job completion time is unexpressible. Explicit resource requirements (e.g., small VM) makes VM packing simpler but since VMs last for much longer than tasks mistakes in packing substantially reduce gains.

7. CONCLUSION

In the context of multi-resource schedulers, we show that the focus on fairness and not allocating IO lead to significantly worse performance. We present heuristics that efficiently pack along multiple resources and prefer jobs with less "remaining work". As a result both cluster efficiency (makespan) and average job completion time improve. Both these aspects are novel in the context of cluster schedulers. Devising an optimal solution to the core problem of scheduling tasks with diverse resource demands is hard; just packing to minimize makespan is APX-Hard. Our approach here is towards finding a practical solution, which requires, in addition to the above, the ability to incorporate fairness. We do not offer new notions of fairness but rather show how achieving desired amounts of fairness can coexist with improving cluster performance through packing. We note that our methods to avoid starvation and properly use the DAG structure of jobs are rather ad-hoc. However, we believe this to be a first substantial step in the right direction. We have implemented our heuristics inside the YARN framework; trace-driven simulations and deployment show encouraging initial results.

Acknowledgements

We thank our shepherd Andrew Moore and the anonymous reviewers for their feedback. Chris Douglas and Carlo Curino offered help and suggestions that significantly improved our prototype. Ganesh was supported in part by NSF grants CCF-1139158, the DARPA XData Award FA8750-12-2-0331 and gifts from various companies. Robert and Aditya were supported in part by NSF grants CNS-1302041, CNS-1314363 and CNS-1040757.

References

[1] Apache Hadoop. `http://hadoop.apache.org`.
[2] Facebook Data Grows By Over 500 TB Daily. `http://bit.ly/1p5EV3c`.
[3] Hadoop MapReduce - Capacity Scheduler. `http://bit.ly/1tGpbDN`.
[4] Hadoop MapReduce - Fair Scheduler. `http://bit.ly/1p7sJ1I`.
[5] Hadoop YARN Project. `http://bit.ly/1iS8xvP`.
[6] Petabyte Storage at Half Price with QFS. `http://bit.ly/1x4A6vF`.
[7] S. Agarwal et al. Re-optimizing data parallel computing. In *NSDI*, 2012.
[8] M. Al-Fares et al. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.
[9] Y. Azar et al. Tight Bounds for Online Vector Bin Packing. In *STOC*, 2013.
[10] M. Chowdhury et al. Managing Data Transfers in Computer Clusters with Orchestra. In *SIGCOMM*, 2011.
[11] M. Chowdhury et al. Leveraging Endpoint Flexibility in Data-Intensive Clusters. In *SIGCOMM*, 2013.
[12] A. Ghodsi et al. Dominant Resource Fairness: Fair Allocation Of Multiple Resource Types. In *NSDI*, 2011.
[13] A. Greenberg et al. A Scalable and Flexible Datacenter Network . In *SIGCOMM*, 2009.
[14] S. Gulwani et al. SPEED: Precise And Efficient Static Estimation Of Program Computational Complexity. In *POPL*, 2009.
[15] C. Guo et al. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers . In *SIGCOMM*, 2009.
[16] M. Harchol-Balter et al. Connection Scheduling in Web Servers. In *USITS*, 1999.
[17] M. Isard et al. Dryad: Distributed Data-Parallel Programs From Sequential Building Blocks. In *EuroSys*, 2007.
[18] M. Isard et al. Quincy: Fair Scheduling For Distributed Computing Clusters. In *SOSP*, 2009.
[19] L. Lu et al. Predictive VM Consolidation on Multiple Resources: Beyond Load Balancing. In *IWQoS*, 2013.
[20] R. Panigrahy et al. Heuristics for Vector Bin Packing. In *MSR TR*, 2011.
[21] A. Rasmussen et al. Themis: An I/O-Efficient MapReduce. In *SoCC*, 2012.
[22] A. Shieh et al. Sharing the Data Center Network. In *NSDI*, 2011.
[23] T. Tannenbaum et al. Condor – A Distributed Job Scheduler. In *Beowulf Cluster Computing with Linux*. MIT Press, 2001.
[24] A. Thusoo et al. Hive: A Warehousing Solution Over A Map-Reduce Framework. *Proc. VLDB Endow.*, 2009.
[25] V. V. Vazirani. Approximation Algorithms. In *Springer-Verlag*, 2001.
[26] A. Wierman et al. Classifying Scheduling Policies with Respect to Unfairness in an M/GI/1 . In *SIGMETRICS*, 2003.
[27] G. J. Woeginger. There Is No Asymptotic Ptas For Two-Dimensional Vector Packing. In *Information Processing Letters*, 1997.
[28] C.-W. Yang et al. Tail Asymptotics For Policies Favoring Short Jobs In A Many-Flows Regime. In *SIGMETRICS*, 2006.
[29] M. Zaharia et al. Delay Scheduling: A Technique For Achieving Locality And Fairness In Cluster Scheduling. In *EuroSys*, 2010.
[30] J. Zhou et al. SCOPE: Parallel Databases Meet MapReduce. *Proc. VLDB Endow.*, 2012.